Recursion, Stratification in SQL3 and Datalog

Lecture #17
Autumn, 2001
Stratified Negation

1. Negation wrapped inside a recursion makes no sense.
2. Even when negation and recursion are separated, there can be ambiguity about what the rules mean, and some one meaning must be selected.
Stratified Negation (II)

1. Stratified negation is an additional restraint on recursive rules (like safety) that solves both problems:
   - 1. It rules out negation wrapped in recursion.
   - 2. When negation is separate from recursion, it yields the intuitively correct meaning of rules.

2. Stratification recently adopted in the SQL3 standard for recursive SQL.
Problem with Recursive Negation

1 Consider:
   \[ P(x) \leftarrow Q(x) \text{ AND NOT } P(x) \]
   \[ Q = EDB = \{1, 2\}. \]

1 Compute IDB \( P \) iteratively?
   » Initially, \( P = \emptyset \).
   » Round 1: \( P = \{1, 2\} \).
   » Round 2: \( P = \emptyset \), etc., etc.
Intuitively: stratum of an IDB predicate = maximum number of negations you can pass through on the way to an EDB predicate.

Must not be ¥ in "stratified" rules.

Define stratum graph:
  » Nodes = IDB predicates.
  » Arc P → Q if Q appears in the body of a rule with head P.
  » Label that arc -- if Q is in a negated subgoal.
Example

1. \[ P(x) \leftarrow Q(x) \text{ AND NOT } P(x) \]

\[ \quad -- \overset{P}{\longrightarrow} \]

©Fall, 2001, LRX
Example

1 Reach(x) <- Source(x)
   Reach(x) <- Reach(y) AND Arc(y,x)
NoReach(x) <- Target(x)
AND NOT Reach(x)

NoReach
  --
Reach
Computing Strata

1. Stratum of an IDB predicate $A = \text{maximum number of -- arcs on any path from } A \text{ in the stratum graph.}$
Examples

» For first example, stratum of P is 1.
» For second example, stratum of Reach is 0; stratum of NoReach is 1.
1. A Datalog program with recursion and negation is stratified if every IDB predicate has a finite stratum.
1. If a Datalog program is stratified, we can compute the relations for the IDB predicates lowest-stratum-first.
Example

1. Reach(x) <- Source(x)
   Reach(x) <- Reach(y) AND Arc(y,x)
   NoReach(x) <- Target(x)
   AND NOT Reach(x)

1. EDB:
   » Source = {1}.
   » Arc = {(1, 2), (3, 4), (4, 3)}.
   » Target = {2, 3}.
Example (II)

1. First compute $\text{Reach} = \{1, 2\}$ (stratum 0).
2. Next compute $\text{NoReach} = \{3\}$. 
Is the Stratified Solution "Obvious"?

1. Not really.
2. There is another model that makes the rules true no matter what values we substitute for the variables.
   - Reach = \{1, 2, 3, 4\}.
   - NoReach = \emptyset.
Is the Stratified Solution "Obvious"? (II)

1. Remember: the only way to make a Datalog rule false is to find values for the variables that make the body true and the head false.

2. For this model, the heads of the rules for Reach are true for all values, and in the rule for NoReach the subgoal NOT Reach(x) assures that the body cannot be true.
SQL3 Recursion

1. WITH
   stuff that looks like Datalog rules
   an SQL query about EDB, IDB

1. Rule =
   [RECURSIVE] R(<arguments>) AS
   SQL query
Example

1. Find Sally's cousins, using EDB Par(child, parent).
Example (II)

WITH
Sib(x,y) AS
    SELECT p1.child, p2.child
    FROM Par p1, Par p2
    WHERE p1.parent = p2.parent AND p1.child <> p2.child,
RECURSIVE Cousin(x,y) AS
    Sib
    UNION
    (SELECT p1.child, p2.child
     FROM Par p1, Par p2, Cousin
     WHERE p1.parent = Cousin.x AND p2.parent = Cousin.y
    )
SELECT y
FROM Cousin
WHERE x = 'Sally';
Plan for Describing Legal SQL3 recursion

1. Define "monotonicity," a property that generalizes "stratification."
2. Generalize stratum graph to apply to SQL queries instead of Datalog rules.
   » (Non)monotonicity replaces NOT in subgoals.
3. Define semantically correct SQL3 recursions in terms of stratum graph.
Monotonicity

1. If relation $P$ is a function of relation $Q$ (and perhaps other things), we say $P$ is monotone in $Q$ if adding tuples to $Q$ cannot cause any tuple of $P$ to be deleted.
Monotonicity Example

1. In addition to certain negations, an aggregation can cause nonmonotonicity.
   Sells(bar, beer, price)
   SELECT AVG(price)
   FROM Sells
   WHERE bar = 'Joe''s Bar';
Monotonicity Example (II)

1. Adding to Sells a tuple that gives a new beer Joe sells will usually change the average price of beer at Joe's.

2. Thus, the former result, which might be a single tuple like (2:78) becomes another single tuple like (2:81), and the old tuple is lost.
Generalizing Stratum Graph to SQL

1. Node for each relation defined by a "rule."
2. Node for each subquery in the "body" of a rule.
Generalizing Stratum Graph to SQL (II)

1. Arc $P \rightarrow Q$ if
   a) $P$ is "head" of a rule, and $Q$ is a relation appearing in the FROM list of the rule (not in the FROM list of a subquery).
   b) $P$ is head of a rule, and $Q$ is a subquery directly used in that rule (not nested within some larger subquery).
   c) $P$ is a subquery, and $Q$ is a relation or subquery used directly within $P$. 
Generalizing Stratum Graph to SQL (III)

1. Label the arc -- if P is not monotone in Q.
2. Requirement for legal SQL3 recursion: finite strata only.
Example

1. For the Sib/Cousin example, there are three nodes: Sib, Cousin, and SQ (the second term of the union in the rule for Cousin).

2. No nonmonotonicity, hence legal.
A Nonmonotonic Example

1. Change the UNION to EXCEPT in the rule for Cousin.

   RECURSIVE Cousin(x,y) AS
   Sib
       EXCEPT
       (SELECT p1.child, p2.child
        FROM Par p1, Par p2, Cousin
        WHERE p1.parent = Cousin.x
        AND p2.parent = Cousin.y
       )
A Nonmonotonic Example (II)

1. Now, adding to the result of the subquery can delete Cousin facts; i.e., Cousin is nonmonotone in SG.

   ![Diagram](image)

1. Infinite number of --'s in cycle, so illegal in SQL3.
Another Example: NOT Doesn't Mean Nonmonotone

1 Leave Cousin as it was, but negate one of the conditions in the where-clause.
   RECURSIVE Cousin(x,y) AS
     Sib
       UNION
       (SELECT p1.child, p2.child
        FROM Par p1, Par p2, Cousin
        WHERE p1.parent = Cousin.x
        AND NOT (p2.parent = Cousin.y)
       )
Another Example: NOT Doesn't Mean Nonmonotone (II)

1. You might think that SG depends negatively on Cousin, but it doesn't.
   » If I add a new tuple to Cousin, all the old tuples still exist and yield whatever tuples in SG they used to yield.
   » In addition, the new Cousin tuple might combine with old p1 and p2 tuples to yield something new.